Structure Elicitation in Database Reverse Engineering[1]

J-L. Hainaut, J. Henrard, D. Roland, V. Englebert, J-M. Hick
Institut d'Informatique, the University of Namur, rue Grandgagnage, 21 - B-5000 Namur
jlh@info.fundp.ac.be

Abstract. Recovering the semantic description of file and database structures is an important aspect of business application reverse engineering. It includes a particularly delicate activity, namely data structure extraction, i.e. finding the exact data structures and integrity constraints of the database. This process is made more complex than generally expected due to the fact that these structures and constraints often are not explicitly defined, but are translated into implicit constructs, controlled and managed through procedural code or user interface protocol for instance. This paper describes the problem of implicit structure elicitation. It proposes an analysis of this phenomenon, and of the techniques and heuristics that can be used in the elicitation process. It develops a set of efficient techniques and a strategy for the elicitation of one of the most common implicit construct, namely the foreign key. The paper also explains how DB-MAIN, a general-purpose database reverse engineering CASE tool, can help analysts elicit implicit constructs, and specifically foreign keys.

1. Introduction

Database reverse engineering (DBRE) is a software engineering process through which one tries to understand and redocument the files and/or the database of an application. This process can be proposed as the first step of the reverse engineering of a whole application, including its processing components and its user interface [18].

More precisely, DBRE is to yield the complete logical schema and the conceptual schema of this database. These schemas are now standard in data abstraction as found in current database methodologies [2, 16]. By *logical schema*, one means the description of the data structures as they are implemented by the Data Manager, and as they are seen by the application programmer. For instance, the logical schema of a relational database describes its tables, columns, primary and foreign keys as well as all the explicit and implicit constraints to which the data are submitted. The *conceptual schema* of a database is an abstract, computer-independent, description of the information that the data implement. A conceptual schema expressed into the Entity-relationship model comprises entity types, relationship types, attributes and various properties and constraints that translate the concepts and structures of the application domain.

The problem of data structure understanding is particularly complex when old, ill-designed and poorly documented applications are addressed. Indeed, practical experiments have shown that operational applications include many non standard, and even awkward, structures [3, 14].

The Database Engineering Research Group of the University of Namur has proposed, since 1989, a general methodology for tackling this problem, and has developed a generic CASE tool to support reverse engineering processes. According to this methodology, the problems that arise in database reverse engineering naturally fall in two categories that are addressed by the two major processes of DBRE, namely *data structure extraction* and *data structure conceptualization*. The *Data structure extraction* process aims at rebuilding a complete logical schema in which all the explicit and implicit structures and properties are documented. The main source of problems is the fact that many constructs and properties are implicit, i.e. they are not explicitly declared, but they are controlled and managed through, say, procedural sections of the programs. Recovering these structures uses *DDL text analysis*, to extract declared data structures, and *data structure elicitation* techniques to find the implicit ones. The *Data structure conceptualization* process tries to specify the semantic structures of this logical schema as a conceptual schema.

A general introduction to the methodology can be found in [6]. The fundamentals of Data structure conceptualization process are described in [7], and CASE support has been developed in [9] then in [11].

This paper is dedicated to the core of the Data structure extraction process, namely *data structure elicitation* techniques.

In this paper, we briefly recall the main aspects of this methodology (Section 2), then we analyze in further detail the data structure elicitation process. In Section 3, we define the concept of implicit construct, and we discuss its technical and behavioural origins. In Section 4, we describe the main information sources and heuristics used in data structure elicitation. Section 5 presents a short description of the thirteen most common implicit constructs. Section 6 develops one of them in detail, namely the *foreign key* that implements inter-file relationships. Finally, we briefly discuss CASE support in Section 7.

2. DBRE Methodology

[1] This research is a part of the DB-MAIN project [8], which is partly supported by the European Community, by the Région Wallonne and by a consortium comprising ACEC-OSI (Be), ARIANE-II (Be), Banque UCL (Lux), BBL (Be), Centre de recherche public H. Tudor (Lux), CGER (Be), Cockerill-Sambre (Be), CONCIS (Fr), D'Ieteren (Be), DIGITAL (Be), EDF (Fr), EPFL (CH), Groupe S (Be), IBM (Be), OBLOG Software (Port), ORIGIN (Be), Ville de Namur (Be), Winterthur (Be), 3 Suisses (Be). The DB-PROCESS subproject is supported by the Communauté Française de Belgique.

The general architecture of the reference DBRE methodology is outlined in Fig. 1. It shows clearly the two main processes that will be described in Sections 2.1 and 2.2. These processes are generic, in that they are independent of the data model and of the programming language used to develop the application. In the following, the term Data Management System (DMS) stands for either File Management Systems or DataBase Management Systems.

2.1 Data Structure Extraction

This phase consists in recovering the complete DMS schema, including all the implicit and explicit structures and constraints. True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL texts, etc). Though essential information may be missing from this schema, the latter is a rich starting point that can be refined through further analysis of the other components of the application (views, sub-schemas, screen and report layouts, procedures, fragments of documentation, database content, program execution, etc).

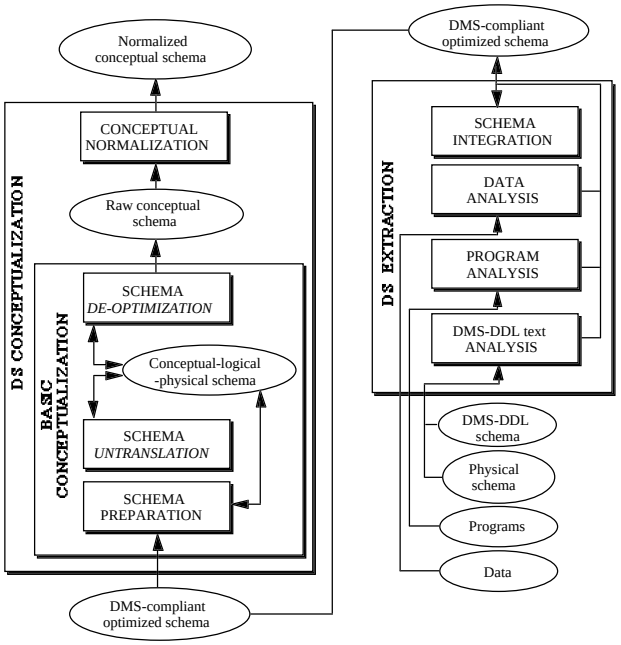

Fig. 1. Main components of the generic DBRE methodology.

The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases. The analysis of each source program provides a partial view of the file and record structures only. For most real-world applications, this analysis must go well beyond the mere detection of the record structures declared in the programs. In this methodology, the main processes of Data Structure Extraction are the following:

- DMS-DDL text ANALYSIS. Consists in analyzing the data structures declaration statements (in the specific DDL) included in the schema scripts and application programs. It produces a first-cut logical schema.
- PROGRAM ANALYSIS. Consists in analyzing the other parts of the application programs, a.o. the procedural sections, in order to detect evidence of additional data structures and integrity constraints.
- DATA ANALYSIS. This refinement process examines the contents of the files and databases in order (1) to detect data structures and properties (e.g. to find the unique fields or the functional dependencies in a file), and (2) to test hypotheses (e.g. "could this field be a foreign key to this file ?").
- SCHEMA INTEGRATION. When more than one information source has been processed, the analyst is provided with several, generally different, extracted (and possibly refined) schemas that must be merged.

In this description, *data structure elicitation* techniques are used in the first three processes. The end product of this phase is the complete logical schema. This schema is expressed according to the specific model of the DMS, and still includes possible optimized constructs, hence its name: the *DMS-compliant optimized schema*, or *DMS schema* for short.

2.2 Data Structure Conceptualization

This second phase addresses the conceptual interpretation of the DMS schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. It comprises two sub-processes, namely *Basic conceptualization* and *Conceptual normalization* [7].

- BASIC CONCEPTUALIZATION. The main objective of this process is to extract all the relevant semantic concepts underlying the logical schema.
- CONCEPTUAL NORMALIZATION. This process restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, e.g. expressiveness, simplicity, minimality, readability, genericity, extensibility [2].

3. Data Structure Elicitation - Problem statement

As explained above, the main problem of the *Data Structure Extraction* phase is to discover, and to make explicit the structures and constraints that were implicitly implemented. In this section we define the concept of implicit construct, and we briefly discuss five situations that generate implicit constructs.

3.1 Explicit *vs* implicit constructs

An explicit construct is a component or a property of a data structure that is declared through a specific DDL statement. An implicit construct is a component or a property that holds in the data structure, but that has not been declared explicitly. Through analysis of the DDL statements alone, the implicit constructs remain undetected. The most popular example certainly is that of *foreign key* (i.e. a field of a record type whose aim is to identify a record in another file). Let us consider the example of Fig. 2, in which two tables, linked by a foreign key, are declared. We can say that this foreign key is an *explicit construct*, insofar as we have used a specific statement to declare it.

```
create table CUSTOMER(C-ID integer primary key,
                      C-DATA char 80)
create table ORDER(   O-ID integer primary key,
                      OWNER integer
           foreign key(OWNER)references CUSTOMER)
```

Fig. 2 - Example of an explicit foreign key.

Fig. 3 represents fragments of an application in which no foreign keys have been declared, but which strongly suggest that column OWNER behaves as a foreign key. If we are convinced that this behaviour must be taken for an absolute rule, then OWNER is an *implicit foreign key*.

```
create table CUSTOMER(C-ID integer primary key,
                      C-DATA char 80)
create table ORDER(   O-ID integer primary key,
                      OWNER integer)
    ...
exec SQL
   select count(*) in :ERR from ORDER
   where OWNER not in (select C-ID from CUSTOMER)
end SQL

if ERR > 0 then display ERR,' ref. violations';
```

Fig. 3 - Example of an implicit foreign key.

3.2 The origin of implicit constructs

By examining the expressive power of DMS, compared with that of semantics representation formalisms, and by analyzing how programmers work, we can identify five major sources of implicit constructs.

Structure hiding. It concerns a source data structure or constraint S1, *which could be implemented in the DMS*. It consists in declaring it as another data structure S2 that is more general and less expressive than S1. In COBOL applications for example, a compound/multivalued field, or a sequence of contiguous fields can be represented as a single-valued atomic field (e.g. a *filler*). In a CODASYL or IMS database, a one-to-many relationship type can be implemented as a many-to-many link, through a record/segment type, or can be implemented as an implicit foreign key. In an SQL-2 database, some referential constraints can be left undeclared as it was common in older SQL-1 schemas (Fig. 3). The origin of stucture hiding is always a decision of the programmer, who tries to meet requirements such as field reusability, genericity, program conciseness, simplicity, efficiency as well as consistency with legacy components of the application.

Generic properties. Some DMS offer general purpose functionalities to enforce a large variety of constraints on the data. For instance, current relational DBMS propose column and table check predicates, views with check option, triggers mechanisms and stored procedures. These powerful techniques can be used to program the validation and the management of complex constraints. The problem is that there is no standard way to cope with these constraints. For instance, constraints such as referential integrity can be encoded in many forms, and their elicitation can prove much more complex than for declared foreign keys.

Non declarative structures. They are structures or constraints *that cannot be declared in the target DMS*, and therefore are represented and checked by other means, such as procedural sections of the application. Most often, the checking sections are not centralized, but are distributed and duplicated in different versions, throughout the application programs. For example, standard files commonly include foreign keys, but current DMS ignore this construct. In the same way, CODASYL DBMS do not provide explicit declaration of one-to-one relationship types, which often are implemented as one-to-many set types + validation procedures.

Environmental properties. In some situations, the environment of the system garantees that the external data to be stored in the database satisfy a given property. Therefore, the developer have found it useless to translate this property in the data structure, nor to enforce it through DBMS or programming techniques. Of course, the elicitation of such constraints cannot be based on data structure and program analysis alone. For example if the content of a sequential file comes from an external source in which uniqueness property is garanteed for one of its field, then the sequential file inherits this property, and an identifier can be defined accordingly.

Lost specifications. They correspond to facts that have been ignored or discarded, intentionally or not, during the development of the system. This phenomenon corresponds to flaws in the system that can translate into corrupted data. However, lost specifications can be undetected environmental properties, in which case the data generally are valid.

4. Sources, Techniques and Heuristics

Though there exist a fairly large set of potentially implicit constructs (Section 5), they can all be elicited from the same information sources and through a limited set of common techniques. Except in small size projects, more than one source will be analyzed to find the data structures and its properties. We will discuss briefly the most common ones.

Current documentation. In most reverse engineering projects, the analyst can rely on some documentation related to the source system. Though these documents often are partial and obsolete, they can bring useful information on the key system components. Of course, the comments that programmers include in the programs can be a rich source of information, provided their reliability can be evaluated.

Technical/physical constructs. There can be some correlation between logical constructs and their technical implementation. For instance, a foreign key often is supported by an index. Therefore, an index can be an evidence that a field could be a foreign key. In most DBMS, technical constructs are defined in specific schemas.

Dataflow analysis. Examining in which variables data values flow in the program can put in light structural or intentional similarities between these variables. For instance, if variable B, with structure Sb receives its values from variable A, with structure Sa, and if Sa is more general than Sb, then A can be given structure Sb. The term *flow* must be taken in a broad

sense: if two variables belong to the same graph fragment, at some time, and in some determined circumstances, then their values can be the same, or one of them can be a direct function of the other. The following table presents some common generating statements:

statement	dataflow graph

`move A to B`
`if A = B then ...`
`proc P(in X:int;out Y:char);`
`P(A,B);`
	A ⟶ B
	A ⟷ B
	A ⟶ X;
	Y ⟶ B

More sophisticated, or less strict relations can be used, such as "if A > B then ..." and "A = B + C". Such patterns do not define equality of values, but rather dependency relations.

Usage pattern analysis. The way data are used, transformed and managed in the program brings essential information on the structural properties of these data. For instance, through the analysis of data validation procedures, the analyst can learn what the valid data values are.

Name analysis. Most programmers try to give programming objects meaningful names. Their interpretation can bring some hints about the meaning of the objects, or about their purpose. In addition, this analysis can detect synonyms (several names for the same object) and homonyms (same name for different objects).

Data analysis. The data themselves can exhibit regular pattern, uniqueness or inclusion properties that provide hint to confirm or disprove structural hypotheses. The analyst can find hints that suggest the presence of identifiers, foreign keys, field decomposition, functional dependencies [4] or that restrict the value domain of a field for instance.

Screen/report layout. A screen form or a structured report can be considered as derived views on the data. The layout of the output data as well as the labels and comments can bring essential information on the data [5].

Program execution. The dynamic behaviour of a program working on the data gives information on the requirements the data have to meet to be recorded in the files, and on links between stored data. In particular, combined with data analysis, populated forms and reports provide a powerful examination means to detect structures and properties of the data.

Domain knowledge. It is unconceivable to start a reverse engineering project without any knowledge on the application domain. Indeed, being provided with an initial mental model of the objectives and of the main concepts of the application, the analyst can consider the existing system as an implementation of this model. The objective is then to refine and to validate this first-cut model.

5. A catalog of representative problems

The variety of implicit constructs can be fairly large [3, 14], even in small projects, and studying the application of the techniques described above to each of them would deserve a full book of impressive size. The space limit of this paper suggests just to mention the main implicit structures and constraints we found in actual DBRE projects of various size and nature [11], and studying one of them, namely *implicit foreign keys*, in further detail to illustrate the proposed methodology. We will briefly describe thirteen of the most common problems we found when recovering the logical schemas of COBOL, SQL (ORACLE, SYBASE), DL/1, IDS-2 and RPG files and databases.

Finding the fine-grained structure of record types and fields. A field, or a full record type, declared as atomic, has an implicit decomposition, or is the concatenation of contiguous independent fields. The problem is to recover the exact structure of this field or of this record type. This pattern is very common in standard file and IMS databases, but it has been found in modern databases as well, for instance in relational tables.

Finding field aggregates. A sequence of seemingly independent fields are originated from a source compound field which was decomposed. The problem is to rebuild this source compound field. This is a typical situation in relational, IMS and TOTAL/IMAGE databases.

Finding multivalued fields. A field, declared as single-valued, appears as the concatenation of the values of a multivalued field. The problem is to detect the repeating structure, and to make the multivalued field explicit. Relational, IMS and TOTAL/IMAGE database commonly include such constructs.

Finding multiple field and record structures. The same field, or record structure, can be used as a mere container for various kinds of value. For instance, a record type appears to contain records of two different types at different times.

Finding record identifiers. The identifier (or unique key) of a record type is not always declared. Such is the case for sequential files for example.

Finding identifiers of multivalued fields. Structured record types often include complex multivalued compound fields. Quite often too, these values have an implicit identifier that must be made explicit.

Finding foreign keys. In multi-file applications, there can be inter-file links, represented by foreign keys, i.e. by fields whose values identify records in another file.

Finding functional dependencies. As commonly recognized in the relational database domain, normalization is a recommended property. However, many actual databases include unnormalized structures, generally to get better performance.

Finding exact minimum cardinality of fields and rel-types. Multivalued fields are generally declared as arrays, whose maximum size is specified by an integer, while the minimum size often is not mentioned, and is under the responsibility of the programmer.

Finding exact maximum cardinality of fields and rel-types. The maximum cardinality can be limited to a specific constant due to implementation constraints. Further analysis can show that this limit is artificial, and represents no intrinsic property of the problem.

Finding redundancies. Very often, actual databases include redundancies that are to provide better performance. It is essential to detect and express them in order to normalize the schema in further reverse engineering steps.

Finding constraints on value domains. In most DBMS, declared data structures are very poor as far as their value domains are concerned. Quite often, though, strong restriction is enforced on the admittable values.

Finding meaningful names. Some programming disciplines, or technical constraints, impose the usage of meaningless names, or of very condensed names whose meaning is unclear. On the contrary, some applications have been developed with no discipline at all, leading to poor and contradictory naming conventions.

6. Implicit foreign key elicitation

6.1 The problem

The foreign key is a field (or combination thereof) whose value is used to reference a record in another (or in the same) file. This construct is a major building block in relational databases, but it has been found in practically all kinds of databases, such as IMS and CODASYL databases, where it is used to avoid the burden of explicit relationships, or to compensate the limits of the DMS. For instance, IMS imposes strong constraints on the number and on the pattern of relationships, while CODASYL DBMSs prohibit cyclic set types. Foreign keys have also been used to ease partitioning CODASYL databases into chunks that need not be on-line simultaneously.

The standard configuration of foreign keys is the following: B.B2 → A.A1, where B2 is a single-valued field (or set of fields) of record type B and A1 is the primary identifier of record type A. B.B2 and A.A1 are defined on the same domain(s). However, practical foreign keys do not always obey the strict recommendations of the relational theory, and richer patterns can be found in actual applications. For instance, we have found a large variety of non standard foreign keys:

- *multivalued* foreign keys: each value of a repeating field is, or includes, a referencing value,
- *secondary* foreign keys: foreign keys that reference secondary identifiers instead of primary ones; these secondary identifiers may even be optional,
- *loosely-matching* foreign keys: the type and length of the foreign key can be different of that of the referenced identifier; for instance, a foreign key of type char(10) can reference an identifier of type char(12), and type numeric(6) can be found to match char(6); in addition the structure can be different: an atomic foreign key can be matched with a compound identifier;
- *multi-target* foreign keys: that reference a record of type A, and/or of type B, and/or of type C,
- *conditional* foreign keys: that reference a foreign record only if a condition is met; if the other cases, the value of the "foreign key" is given another interpretation,
- *overlapping* foreign keys: two foreign keys share a common field (e.g. {X,Y} and {Y,Z}),
- *embedded* foreign keys: a foreign key includes another foreign key (e.g. {X,Y} and {Y}),
- *transitive* foreign keys: a foreign key is the (mathematical) composition of two foreigns keys (e.g. C.C3 → B.B1; B.B2 → A.A1; C.C3 → A.A1).

Let us base the discussion on the schema of Fig. 4, in which two record types (or *tables*, or *segment types*) CUSTOMER and ORDER *may* be linked by a foreign key. We assume that CID is the identifier of CUSTOMER, and that, should a foreign key exist in ORDER, it references this identifier (in short, we do not have to elicit the target identifier).

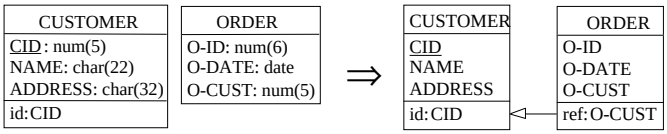

Fig. 4 - Foreign key elicitation - Source and final schemas.

We will examine in which way each of the techniques and heuristics described in Section 4 can be used to elicit foreign key O-CUST, that is to collect hints and evidence contributing to prove, or disprove that field O-CUST is a foreign key to record type CUSTOMER. Afterwards, we propose a tentative strategy to find implicit foreign keys in relational databases.

Though we will discuss the problem of proving that a definite field is a foreign key, it must be noted that this problem may appear in several variants which can be solved by generalizing the techniques that will be examined below. For instance, we could try to find

- all the record types referenced by foreign key O-CUST,
- all the foreign keys in record type ORDER,
- all the foreign keys that reference CUSTOMER.

6.2 Current documentation

When it still exists, and when it can be relied on, the documentation is the first information source to use. Normally, file structures, field description, and particularly their roles (such as referencing) should be documented. Some weaker, but probably more up-to-date, information could be found in the system data dictionary. Indeed, most RDBMS allow administrators to add a short comment to each schema object. The programs themselves should include, through *comments*, information on critical components, such as data structures or validation procedures.

6.3 Physical structure

A foreign key is a mechanism that implements links between records, and is the privileged way to represent inter-entity relationships. We can assume with little risk that application programs will navigate among records following these relationships. Therefore, most foreign keys will be supported by some access mechanisms, such as **indexes**. *Heuristics*: a field supported by an index could be a foreign key, specially when it is not an identifier (most foreign keys implement many-to-one links).

Quite naturally, the candidate field should have the **same domain** of value, i.e. the same type and length, as the identifier of the referenced record type. However, some

matching distorsions can be found as far as lengths and even types are concerned. *Heuristics*: the candidate foreign key must match, strictly or loosely, an identifier of the candidate referenced record type.

In some RDBMS **clustering** mechanisms are proposed to group in the same pages records that have some kind of logical relationships. Joins based on primary-key/foreign-keys are the most common logical relationship. *Heuristics*: if a cluster gathers the records of table A through column A1 and of table B through column B1, and if A1 is an identifier of A, then B1 could be a foreign key referencing A.

In DMS where records can be read in **sorted sequence**, such as in standard file organizations, or in RDBMS (with *order by* clause), it is common practice to interleave records of different types in such a way that they are read in a hierarchical arrangement. Fig. 5 represents in an abstract way a file comprising ORDER and DETAIL records, sorted on common field ID. Field ID is a global identifier for records of both types (hence the disjoint constraint that states that no ORDER.ID value can be a DETAIL.ID value). The identifiers (ID) of both types have the same format, and are made of two components. The second component of all ORDER records contains a *low-value* constant. In this way, each ORDER record is followed by a sequence of DETAIL records that have the same O-ID value. Obviously, DETAIL.ID.O-ID is a foreign key to ORDER, while the *filler* field bears no semantics, and can be ignored. *Heuristics*: in structures such as that described above, the first component of the second record type can be a foreign key to the first record type; in addition, the second component of the first record type can be discarded.

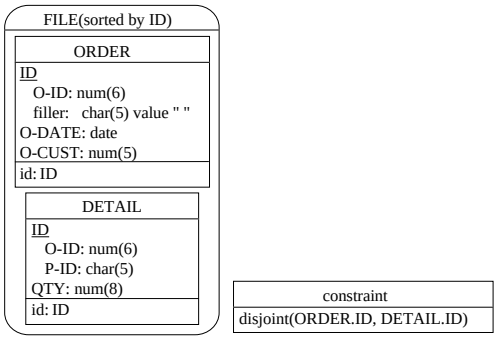

Fig. 5 - Hierarchically sequenced records

6.4 Dataflow analysis

If a foreign key holds between two record types, then we should find, in some programs, data values *flowing* between variables that represent the foreign key and the target identifier. Considering equality relations only, such as those defined in the table of section 4, extracted from the program of Fig. 6, we compute the *equality* dataflow graph of Fig. 7. It shows that, at given time points, CUSTOMER.CID and ORDER.O-CUST share the same value. It is reasonable to think that the same property holds in the files themselves.

```
DATA DIVISION.                PROCEDURE DIVISION.
FILE SECTION.                   ...
FD F-CUSTOMER.                  display "Enter order number"
01 CUSTOMER.                        with no advancing.
  02 CID pic 9(5).              accept CID.
  02 NAME pic X(22).            move 0 to IND.
  02 ADDRESS pic X(32).         call "SET-FILE" using C,IND.
FD F-ORDER.                     read F-ORDER
01 ORDER.                           invalid key go to ERR1.
  02 O-ID pic 9(6).             ...
  02 O-DATE PIC 9(8).           if IND > 0 then
  02 O-CUST pic 9(5).           move O-CUST of ORDER to C.
WORKING-STORAGE SECTION.            ...
  01 C pic 9(5).                if C = CID of CUSTOMER then
  ...                           read F-CUSTOMER
                                   invalid key go to ERR2.
                                      ...
```

Fig. 6 - Excerpts from a program working on CUSTOMER and ORDER

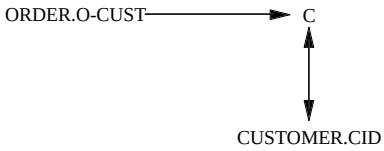

Fig. 7 - Fragment of the equality dataflow graph of the program of Fig. 6.

6.5 Usage pattern analysis

While the dataflow graph only provides us with an abstraction of the relationships between fields, we could be interested by the real thing, i.e. by the procedure that processes records and fields. Let us consider the excerpt of the program of Fig. 6 that is presented in Fig. 8(a). A simpler but equivalent version is proposed in (b).

```
read F-ORDER.                   read F-ORDER.
  invalid key go to ERR1.       move O-CUST of ORDER
move O-CUST of ORDER to C.            to CID of CUSTOMER.
if C = CID of CUSTOMER          read F-CUSTOMER.
  then read F-CUSTOMER
  invalid key go to ERR2.
          (a)                             (b)
```

Fig. 8 - (a) excerpts from the program of Fig. 6. (b) a simplified equivalent version.

The latter exhibits a common *cliché* of file processing programs: reading a record (CUSTOMER) identified by a field value (O-CUST) from another record (ORDER). This is a typical way of navigating from record to record, that can be called *procedural join*, by analogy with the relational join of SQL-based DBMS. The probability of O-CUST being a foreign key to CUSTOMER is fairly high.

Let us consider the example of foreign key of Fig. 9.

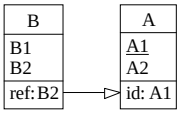

Fig. 9 - An elementary abstract schema.

The main processing patterns that use the instances of this schema are summarized in Fig. 10. Both the predicative (SQL) and the procedural (pseudo-code) versions are given.

Function	Procedural pattern

Find the A
of a given B
	`read A(A1=B.B2);`
	`if not found`
	`  then error end-if;`

Find the Bs
of a given A
	`read-first B(B2=A.A1);`
	`while found do process B;`
	`  read-next B(B2=A.A1)`
	`end-while;`

Create a B record	`read A(A1=B.B2);`
	`if found`
	`  then create B end-if;`

Delete an A record	`read-first B(B2=A.A1);`
	`while found do delete B;`
	`  read-next B(B2=A.A1)`
	`end-while;`
	`delete A;`

Function	**SQL-like expressions**

Find the A
of a given B
	`1)select * from A where A1`
	`  in (select B2 from B where ..)`
	`2)select A1,A2 from A,B`
	`  where A.A1=B.B2`

Find the Bs
of a given A
	`1)select * from B where B2`
	`  in (select A1 from A where ..)`
	`2)select B1,B2 from A,B`
	`  where A.A1=B.B2`

Create a B record	`if exists (select *`
	`  from A where A1=B.B2)`
	`then insert into B values(..)`

Delete an A record	`delete from B where B2 in`
	`  (select A1 from A where ..)`
	`delete A where ...`

Fig. 10 - Main processing patterns related to foreign keys.

Applied on the example used so far, these patterns could instantiate into the following code sections:

SQL query	CODASYL DML query

`select CID,NAME,O-DATE`
`from CUSTOMER,ORDER`
`where CID = O-CUST`
	`move O-CUST of ORDER`
	`  to CID of CUSTOMER.`
	`read CUSTOMER record.`

Fig. 10 gives only a few popular expressions of the standard processing functions. Many other variants exist. For instance, SQL offers several ways to code referential integrity implicitly: through *check predicates*, through *SQL triggers*, through *dialog triggers* (e.g. ORACLE SQL-Forms), through *stored procedures* or through *views with check option*. In addition, each technique allows the developer to express the constraint validation in his/her own way. Several authors have proposed heuristics to detect foreign keys by usage pattern analysis [1, 13, 15].

6.6 Name analysis

Quite often, the name of a foreign key suggests its function and/or its target. The names can be less informative for multi-component foreign keys. Sometimes, a common prefix may give useful hints. In our example, the name O-CUST includes a significant part of the name of the target record type. The following rules are among the most common encountered in practice:

- the name suggests the referencing function: CUST-**REF**
- the name suggests the referenced record type or a synonym: CUST, CUSTOMER, CLIENT
- the name suggests the referenced identifier: CID, CUST-ID, C-CODE
- the name suggests the role of the referenced record type: OWNER.
- the name suggests the semantic link implemented by the foreign key: PLACED-BY.

6.7 Domain knowledge

Everybody knows that *customers place orders.* Obviously, record types CUSTOMER and ORDER should be linked in some way. The question is: how ?

6.8 Data analysis

If O-CUST is a foreign key to CUSTOMER, then referential integrity should be satisfied, and each of its values must identify a CUSTOMER record. A small program, or the following SQL query will check this condition:

```
select 'Violations: ',count(*)
from ORDER
where O-CUST not in (select CID from CUSTOMER)
```

However the result n returned by this query must be interpreted with much caution, because several conclusions can be drawn from it:

- n = 0 ⇒ O-CUST is a foreign key,
  ⇒ statistical accident; tomorrow, the result may be different.
- 0 < n < ε ⇒ O-CUST is not a foreign key,
  ⇒ the query has detected data errors,
  ⇒ O-CUST is a conditional foreign key.
- 0 << n ⇒ O-CUST is not a foreign key,
  ⇒ O-CUST is a conditional foreign key.

Some authors [Petit,94] have pushed further the analysis of data inclusion properties.

6.9 Screen/report layout

Screen forms are used to present data and/or to let users enter them. Frequently, one screen panel includes data from several record types. Typically, an order-entry panel will comprise fields whose contents will be dispatched to ORDER, CUSTOMER, DETAIL and PRODUCT records. Three kinds of information can be derived from the examination of such forms:

- *Spatial relationships between data fields.* The way the data are located in the screen may suggest implicit relationships.
- *Labels and comments included in the panel*. They bring information on the meaning, the role and the constraints of each screen field.
- *Missing fields.* If field O-CUST does not appear on the screen, then it may be useless to record its value. This may mean that this field designates an information that is given by the context, for instance about the customer of the order.

A screen layout can be examined as a standalone component, as suggested above. It can also be analyzed as source/target data structures of the programs that use it to communicate with their environment. Fig. 11, shows how screen data are

distributed in the data files. An implicit join based on CUSTOMER.CID = ORDER.O-CUST is clearly suggested. Including screen fields in the dataflow graph is another way to make these links explicit.

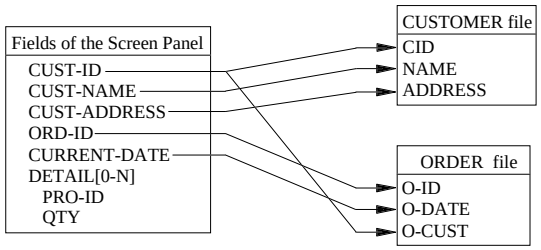

Fig. 11 - Detecting a foreign key in a screen panel.

Data reports can be considered both as data structures and as populated views of the persistent data. The first aspect is quite similar to that of screen layout: a report is a hierarchical data structure that makes relationships between data explicit. The second one relates to the data analysis heuristics.

6.10 Program execution

The principle is to analyze the reactions of the program to selected stimuli, for instance in terms of acceptance and rejection of input data and update queries. If the program rejects a tentative data entry concerning an order unless its O-CUST value appears as the CID value of some CUSTOMER record, then we can conclude that the program enforces some kind of inclusion property between these value sets, which can be interpreted as referential integrity. Similarly, if the program refuses to delete a CUSTOMER record because the customer still has pending orders, we can translate this behaviour into the fact that ORDER records depend on this CUSTOMER record.

A running program also populates the screen panels, just like printed reports. New relationships can be detected in this way.

6.11 Tentative strategy

Trying to build a general strategy for foreign key elicitation that would be valid in any circumstance would be unrealistic. Indeed, we have demonstrated [9] that database reverse engineering basically is a loosely structured learning process which varies largely from one project to another. Nevertheless we can sketch the following principles, based on the schema of Fig. 9, that can apply on relational databases managed by early RDBMS, in which no keys were explicitly declared.

Phase	Heuristics	Short description
Hypothesis Triggering	name analysis	name of column B.B2 suggests a table, or an external id, or includes keywords such as *ref*, ...
	domain knowledge	objects described by B are known to have some relation with those described by A
Hypothesis Completion	name analysis	select table A based on the name of B2
	domain knowledge	find a table describing objects which are known to have some relation with those described by B
	technical constructs	search the schema for a candidate referenced table and id (with same type and length)
Hypothesis	*$Y \equiv B.B2 \rightarrow A.A1$*	*$Y \equiv$ "field B2 of B is a foreign key to identifier A1 of A"*
Hypothesis Proving	technical constructs	there is an index on B2
	technical constructs	B.2 and A.A1 are in the same cluster
	dataflow analysis	B.2 and A.A1 are in the same dataflow graph fragment
	usage pattern	A.A1 values are used to select B rows with same B2 values
	usage pattern	B.2 values are used to select A row with same A1 value
	usage pattern	a B row is stored only if there is a matching A row
	usage pattern	when an A row is deleted, B rows with B2 values equal to A.A1 are deleted as well
	usage pattern	there are *views* based on a join with B.B2 = A.A1
	usage pattern	there is a *view with check option* selecting Bs which match A rows
Hypothesis Disproving	data analysis	find some B.B2 values that are not in A.A1 value set;
	usage pattern	find sequences that store B rows without requiring a matching A row;
	usage pattern	find sequences that delete an A row without deleting matching B rows;

Fig. 12 - Tentative 4-phase strategy for detecting implicit foreign keys in early relational databases. The first phase uses initial hints that trigger the analysis. The second phase consists in completing the hypothesis. Once the latter is stated, the analyst uses positive techniques, that tend to prove the hypothesis, and/or negative techniques, that tend to disprove the hypothesis.

7. CASE support

The processes and the heuristics proposed in this paper require powerful tools to help the analysts in processing huge amounts of complex program texts, documentation and data. The DB-MAIN CASE tool developed by the Database Engineering Research Group of the University of Namur includes several processors that are intended to these tasks. The first version of DB-MAIN has been presented in [9, 11], so that we will only mention in this paper some of the components dedicated to implicit structure elicitation.

7.1 General functions

DB-MAIN is a general-purpose programmable CASE environment dedicated to *Database Application Engineering*. In particular, it is to help developers and analysts in the development, re-engineering, migration and evolution of data-centered applications [8]. DB-MAIN offers the functions that are now standard in most commercial CASE tools: specifications management, evaluation, conceptual/logical translation, graphical viewing, report and code generation. However, it also includes many original functions that are required to support non standard activities such as reverse engineering, optimization, customized code generation, migration, etc.

An important aspect of DB-MAIN is its functional extensibility. For instance, a powerful repository-based programming language, *Voyager 2*, has been associated with the tool in order to allow analysts to develop their own CASE processors. This language makes it very easy to develop specific generators, parsers, evaluators, checkers, transformations or report writers.

As far as specific reverse engineering aspects are to be supported, DB-MAIN includes processors such as code extractors (currently for COBOL, RPG, IMS, SQL, IDS-2), reverse transformation toolkits [7], text browsers, schema viewers, schema evaluators, schema integrators, name analyzers and report generators to mention few of them.

In the limited scope of this paper, we describe some of the DB-MAIN assistants dedicated to implicit construct elicitation only.

7.2 The assistants

DB-MAIN offers a collection of assistants, some of which are dedicated to reverse engineering activities. An assistant is a sort of expert in a specific kind of tasks, or in a class of problems, intended to help the analyst in frequent, tedious or complex activities. Among the assistants specifically intended to implicit construct elicitation, we will briefly describe the *Text analysis* and *Reference key finding* assistants.

7.2.1 The Text analysis assistant

This assistant provides a set of sophisticated tools to browse texts such as program source files, to search them for complex text patterns, and to compute abstractions such as dataflow graphs and call graphs. We briefly describe three processors provided by this assistant.

Interactive pattern-matching engine. The *pattern-matching* engine searches text files for definite patterns or *clichés* expressed in PDL, a *Pattern Definition Language*. This is the main tool to perform *usage patterns* analysis.

Dataflow graph builder and inspector. This tool is parametrized with a PDL patterns library that defines the selected relationships between program variables. The analyst can select a variable A, then examine in context the statements that mention any variable that is related to A, directly or transitively.

Program slicer. This processor builds the program slice [17] relative to a program point. The program slice can be visualized in context, displayed in selected color, or extracted as an autonomous program on which other tools can be applied, such the pattern-matching engine, the dataflow builder or the program slicer itself.

7.2.2 The Reference key finding *assistant*

This DMS-independent tool proposes some popular heuristics to find foreign keys (as well as more general *inclusion* and *copy* constraints) in any schema, whatever its level of abstraction and the data model in which it is expressed. Below, we define its most significant aspects.

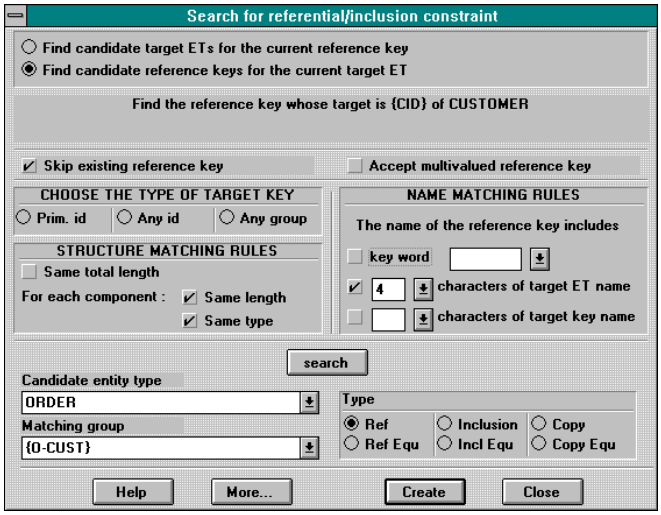

Fig. 13 - The Foreign key searching Assistant.

Two strategies are available: *(1)* given a candidate foreign key, find the possible target record types, *(2)* given a record type R (and one of its identifier), find the fields of the schema that could reference R (the potential foreign keys).

The rules governing the way foreign keys are compared with target keys can be customized. Two sets of rules are proposed. *Structure matching rules*: either their overall length (sum of the lengths of the components) must be the same, or they must have the same structure (the components have same length and/or same type when compared pairwise). *Name matching rules*: (1) the name of the candidate foreign key includes a selected key word (e.g. *ref*, *id*, *code*, *oid*, etc), (2) the name of the candidate foreign key (or of its components) includes a certain number of characters of the name of the target record type, (3) the name of the candidate foreign key (or of its components) includes a certain number of characters of the name of the identifier of the target record type

Once the desired searching rules are set, the elicitation process can be performed (*search* button). The assistant searches the current schema, and proposes a list of candidate record types. For each of them, it proposes the matching groups of fields. Finally, the analyst selects the best candidate Fig. 13 illustrates the use of the assistant in the following scenario:

- strategy 2: the analyst selects the identifier {CID} of record type CUSTOMER, then asks the assistant to find the possible foreign keys;

- only unknown and single-valued keys are searched for;
- the keys match if they have same type and length;
- they also match if, in addition, the name of the foreign key includes at least 4 characters of the target record type name (i.e. CUSTOMER);
- the analyst selects the field O-CUST in suggested record type ORDER, and creates a standard foreign key.

It is interesting to note that, at far as the authors know, no commercial CASE tools offer even elementary help in construct elicitation. As an example, most tools that propose heuristics to detect foreign keys use primitive built-in rules such as the following: *if a field has the same name as the identifier of record type R, then it is declared a foreign key to R.* It is rather simple to prove that this rule will ignore most foreign keys, and that some fields which happen to have the same name will be unduly declared foreign keys.

The result of this weakness is well known by practitioners: providing the analysts with practically no support in elicitation, the current tools build an incomplete logical schema, which will be translated into an incomplete conceptual schema. Hence the very low level of penetration, and the even lower user's satisfaction rate, of current Reverse Engineering CASE tools[2]. As an evidence, it is interesting to observe that many CASE tools once claiming to offer DBRE capabilities either are disappearing, or discard this argument from their commercial presentation.

8. Conclusions

For several years, several authors have shown that, due to non standard programming styles and to technological limits of the Data Management Systems, a significant part of the structural information on the data can be deeply burried in the procedural code of the programs, in the user interface, and in the data themselves.

In this paper, we have analyzed this phenomenon, its possible origins and the main elicitation techniques and heuristics to extract this structural information, a process called *implicit constructs elicitation*. Among the important implicit constructs, we have chosen one of the most representative, namely the foreign key, and we have proposed a flexible elicitation strategy. In addition, we have developed in the DB-MAIN CASE tool generic and specific processors and assistants dedicated to construct elicitation, and more specifically to foreign key elicitation.

This analytical work and CASE support are being extended to the other implicit constructs (a draft can be found in [10]).

9. References

[1] Andersson, M., Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proc. of the 13th Int. Conf. on ERA*, Springer-Verlag, 1994.

[2] Batini, C., Ceri, S., Navathe, S., *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/ Cummings, 1992.

[3] Blaha, M.R., Premerlani, W., J., Observed Idiosyncracies of Relational Database designs, in [18], 1995.

[4] Bitton, D., Millman, J., Torgersen, S., A Feasibility and Performance Study of Dependency Inference, in *Proc. IEEE Data Engineering Conference*, Los Angeles, 1989.

[5] Choobineh, J. et al., An Expert Database Design System based on Analysis of Forms, IEEE Tr. Soft. Engineering 4:2, 1988

[6] Hainaut, J-L., Chandelon M., Tonneau C., Joris M., Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE WCRE*, May 1993, IEEE CS Press, 1993

[7] Hainaut, J-L, Chandelon M., Tonneau C., Joris M., Transformational techniques for database reverse engineering, in *Proc. of the 12th Int. Conf. on ERA*, E/R Institute and Springer-Verlag, LNCS, 1993

[8] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D. Evolution of database Applications: the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ERA*, Springer-Verlag, 1994.

[9] Hainaut, J-L., Englebert, V., Henrard, J., Hick J-M., Roland, D., Requirements for Information System Reverse Engineering Support, in [18], 1995

[10] Hainaut, J-L., *Database Reverse Engineering - Problems, Methods and Tools*,Tutorial notes, CAiSE•95, Jyväskylä, Finland, May. 1995 (available at jlh@info.fundp.ac.be)

[11] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V., Database Reverse Engineering: from Requirements to CARE tools, *J. of Automated Soft. Engineering*, 3:2, 1996

[12] Joris, M., Van Hoe, R., Hainaut, J-L., Chandelon M., Tonneau C., Bodart F. et al. *PHENIX: methods and tools for database reverse engineering*, in Proc. 5th Int. Conf. on Software Engineering and Applications, December 1992, EC2, 1992

[13] Petit, J-M., Kouloumdjian, J., Bouliaut, J-F., Toumani, F., Using Queries to Improve Database Reverse Engineering, in *Proc. of the 13th Int. Conf. on ERA*, Manchester, Springer-Verlag, 1994

[14] Premerlani, W., J., Blaha, M.R. An Approach for Reverse Engineering of Relational Databases, in *Proc. of the IEEE WCRE*, IEEE CS Press, 1993

[15] Signore, O, Loffredo, M., Gregori, M., Cima, M. Reconstruction of ER Schema from Database Applications: a Cognitive Approach, in *Proc. of the 13th Int. Conf. on ERA*, Manchester, Springer-Verlag, 1994.

[16] Teorey, T. J., *Database Modeling and Design : the Fundamental Principles*, Morgan Kaufman, 1994.

[17] Weiser, M., Program Slicing, *IEEE TSE*, Vol. 10, 1984.

[18] Wills, L., Newcomb, P., Chikofsky, E., (Eds), *Proc. of the 2nd IEEE WCRE*, Toronto, July 1995, IEEE CS Press, 1995.

[2] "*I am very dissatisfied with the capabilities of commercial CASE tools for DBRE. The best do little more than save keystrokes for entering the raw schema*" [anonymous referee of this paper].